

FOUNDATION-0.11 PART 1: Prometheus Monitoring Setup

OptiInfra Development Series

Phase: Foundation (Week 1)

Component: Monitoring Infrastructure - Prometheus

Estimated Time: 15 minutes setup + 10 minutes validation

Dependencies: P-01 (Bootstrap), 0.10 (Shared Utilities)

Overview

This prompt creates a complete Prometheus monitoring setup for OptiInfra, including:

- Prometheus server configuration
 - Service discovery for all components
 - Custom metrics exporters for each agent
 - Alert rules for critical conditions
 - Integration with existing services
-

Objectives

By the end of this prompt, you will have:

1. Prometheus server running and scraping all services
 2. Custom metrics exposed by orchestrator and agents
 3. Alert rules configured for critical conditions
 4. Service discovery working for dynamic targets
 5. Metrics accessible via Prometheus API
-

Prerequisites

Before starting, ensure:

- P-01 (Bootstrap) completed - all services structure exists
 - 0.10 (Shared Utilities) completed - logging and config utils available
 - Docker and Docker Compose installed
 - Ports 9090 (Prometheus), 8080-8004 (services) available
 - All services from previous prompts are running
-



Detailed Windsurf Prompt



Create a complete Prometheus monitoring setup for OptiInfra multi-agent system.

CONTEXT:

- Multi-agent LLM infrastructure optimization platform
- 4 agents (Cost, Performance, Resource, Application) + 1 orchestrator
- Services: orchestrator (Go:8080), cost-agent (Python:8001), performance-agent (8002), resource-agent (8003), application-agent (8004)
- Existing infrastructure: PostgreSQL, ClickHouse, Qdrant, Redis
- Project root: /optiinfra

REQUIREMENTS:

1. PROMETHEUS CONFIGURATION (monitoring/prometheus/)

Create prometheus.yml with:

- Global config: scrape_interval: 15s, evaluation_interval: 15s
- Scrape configs for ALL services:
 - * orchestrator (Go): http://orchestrator:8080/metrics
 - * cost-agent (Python): http://cost-agent:8001/metrics
 - * performance-agent: http://performance-agent:8002/metrics
 - * resource-agent: http://resource-agent:8003/metrics
 - * application-agent: http://application-agent:8004/metrics
 - * PostgreSQL: postgres-exporter:9187
 - * ClickHouse: clickhouse-exporter:9116
 - * Redis: redis-exporter:9121
- Service discovery via Docker DNS
- Metric relabeling for consistent naming

2. ALERT RULES (monitoring/prometheus/alerts.yml)

Create rules for:

- Service health: service_up == 0 for >1m
- High latency: request_duration_seconds > 1.0 (p95)
- Error rate: error_rate > 0.05 (5%)
- Database connections: connection_pool_usage > 0.9
- Agent failures: agent_execution_failures > 3 per 5m
- Cost anomalies: cost_delta > 0.2 (20% unexpected increase)
- Quality regressions: quality_score < 0.95
- Resource exhaustion: gpu_utilization > 0.95 or memory_usage > 0.9

3. ORCHESTRATOR METRICS (services/orchestrator/internal/metrics/)

Add to Go orchestrator:

- prometheus/client_golang integration

- Custom metrics:
 - * agent_requests_total (counter) - labels: agent, status
 - * agent_request_duration_seconds (histogram) - labels: agent
 - * agent_health_status (gauge) - labels: agent
 - * coordination_conflicts_total (counter)
 - * approval_workflow_duration_seconds (histogram)
 - * active_optimizations (gauge)
- /metrics HTTP endpoint handler
- Middleware for automatic request tracking

4. PYTHON AGENT METRICS (shared/python/utils/metrics.py)

Create shared Python metrics module:

- prometheus_client integration
- Base metrics class for all agents:
 - * request_duration_seconds (histogram)
 - * requests_total (counter) - labels: endpoint, status
 - * errors_total (counter) - labels: error_type
 - * llm_api_calls_total (counter) - labels: provider, model
 - * llm_token_usage_total (counter) - labels: provider, type
 - * optimization_executions_total (counter) - labels: type, outcome
 - * recommendation_confidence (histogram)
- FastAPI middleware for automatic tracking
- Custom metric decorators (@track_duration, @track_errors)

5. AGENT-SPECIFIC METRICS

Cost Agent (services/cost-agent/src/metrics.py):

- cost_savings_total (counter) - actual savings in USD
- cost_recommendations_total (counter) - labels: type
- spot_migration_success_rate (gauge)
- reserved_instance_coverage (gauge)

Performance Agent (services/performance-agent/src/metrics.py):

- latency_improvement_ratio (gauge)
- throughput_qps (gauge)
- kv_cache_hit_rate (gauge)
- quantization_speedup (gauge)

Resource Agent (services/resource-agent/src/metrics.py):

- gpu_utilization (gauge) - labels: gpu_id
- scaling_events_total (counter) - labels: direction
- resource_consolidation_ratio (gauge)

Application Agent (services/application-agent/src/metrics.py):

- quality_score (gauge) - labels: metric_type
- regression_detections_total (counter)
- ab_test_results (gauge) - labels: variant, metric

6. DATABASE EXPORTERS (monitoring/exporters/)

Add docker-compose entries:

- postgres-exporter: wrouesnel/postgres_exporter
 - * Connection to PostgreSQL
 - * Custom queries for OptiInfra-specific metrics
- clickhouse-exporter: flyegor/clickhouse-exporter
 - * Connection to ClickHouse
 - * Table sizes, query performance
- redis-exporter: oliver006/redis_exporter
 - * Connection to Redis
 - * Memory usage, key statistics

7. DOCKER COMPOSE UPDATES (docker-compose.yml)

Add services:

- prometheus:
 - * Image: prom/prometheus:latest
 - * Volumes: ./monitoring/prometheus:/etc/prometheus
 - * Ports: 9090:9090
 - * Networks: optiinfra-network
- postgres-exporter, clickhouse-exporter, redis-exporter
- Update all service configs to expose metrics ports

8. TESTING & VALIDATION

Create monitoring/tests/test_prometheus.py:

- Test Prometheus can scrape all targets
- Test alert rules syntax is valid
- Test custom metrics are registered
- Test metric values update correctly
- Test exporters connect to databases

TECHNICAL SPECIFICATIONS:

- Go: Use prometheus/client_golang v1.17+
- Python: Use prometheus-client v0.19+
- Prometheus: Use prom/prometheus:v2.48+
- Metric naming: snake_case, units in name (e.g., _seconds, _bytes)

- Labels: Consistent across services (service, agent, status)
- Histograms: Buckets appropriate for latency (0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1, 2.5, 5, 10)

FILE STRUCTURE:

```

monitoring/
  └── prometheus/
    ├── prometheus.yml      # Main config
    ├── alerts.yml          # Alert rules
    └── queries/            # Sample PromQL queries
  └── exporters/
    └── postgres/
      └── queries.yml      # Custom PostgreSQL queries
    └── clickhouse/
      └── queries.yml      # Custom ClickHouse queries
  └── tests/
    └── test_prometheus.py
services/orchestrator/internal/metrics/
  └── metrics.go          # Prometheus metrics
  └── middleware.go       # HTTP middleware
shared/python/utils/
  ├── metrics.py          # Base metrics class
  └── decorators.py       # Metric decorators
services/cost-agent/src/
  ├── metrics.py          # Cost-specific metrics
  └── main.py              # Updated with metrics endpoint
services/performance-agent/src/metrics.py
services/resource-agent/src/metrics.py
services/application-agent/src/metrics.py

```

IMPLEMENTATION NOTES:

1. Start with `prometheus.yml` configuration
2. Add Go metrics to orchestrator first (simpler)
3. Create Python base metrics class
4. Add agent-specific metrics
5. Add database exporters
6. Update `docker-compose.yml`
7. Test all targets are discovered
8. Validate metrics are being scraped

BEST PRACTICES:

- Use consistent label names across all services

- Include service version in metrics (version label)
- Use counters for totals, gauges for current values, histograms for distributions
- Avoid high-cardinality labels (e.g., user_id, request_id)
- Add help text to all custom metrics
- Use _total suffix for counters
- Use _seconds suffix for durations
- Expose metrics on /metrics endpoint (standard)
- Include both infrastructure and business metrics

ERROR HANDLING:

- Gracefully handle metrics collection failures (don't crash service)
- Log metrics collection errors
- Use default values if metric calculation fails
- Validate metric values before recording (non-negative, within bounds)

VALIDATION STEPS:

After implementation, verify:

1. curl http://localhost:9090/targets - all targets should be "UP"
2. curl http://localhost:8080/metrics - orchestrator metrics visible
3. curl http://localhost:8001/metrics - cost agent metrics visible
4. Open http://localhost:9090 - Prometheus UI loads
5. Query: up{job="orchestrator"} - should return 1
6. Query: agent_requests_total - should show counter data
7. Check alerts: http://localhost:9090/alerts - rules loaded
8. Run: make test-monitoring - all tests pass

Generate complete, production-ready code with:

- All configuration files
- All Go metrics code with proper imports
- All Python metrics code with proper error handling
- Complete docker-compose updates
- Comprehensive tests
- README with setup instructions

Success Criteria

After completing this prompt, verify:

1. Prometheus Server



bash

```
# Check Prometheus is running
curl http://localhost:9090/-/healthy
# Expected: Prometheus is Healthy.

# Check targets
curl http://localhost:9090/api/v1/targets | jq '.data.activeTargets[] | {job: .labels.job, health: .health}'
# Expected: All services showing "up"
```

2. Orchestrator Metrics



bash

```
# Check orchestrator metrics endpoint
curl http://localhost:8080/metrics | grep agent_requests_total
# Expected: agent_requests_total{agent="cost",status="success"} 0

# Trigger a request and check again
curl -X POST http://localhost:8080/api/v1/agents/cost/optimize
curl http://localhost:8080/metrics | grep agent_requests_total
# Expected: Counter incremented
```

3. Python Agent Metrics



bash

```
# Check cost agent metrics
curl http://localhost:8001/metrics | grep cost_savings_total
# Expected: cost_savings_total 0.0

# Check shared metrics
curl http://localhost:8001/metrics | grep requests_total
# Expected: requests_total{endpoint="/health",status="200"} 1
```

4. Alert Rules



bash

```
# Check alert rules loaded
curl http://localhost:9090/api/v1/rules | jq '.data.groups[].name'
# Expected: ["optiinfra_alerts"]

# Check specific alert
curl http://localhost:9090/api/v1/alerts | jq '.data.alerts[] | {alert: .labels.alertname, state: .state}'
```

5. Database Exporters



bash

```
# Check PostgreSQL exporter
curl http://localhost:9187/metrics | grep pg_up
# Expected: pg_up 1

# Check ClickHouse exporter
curl http://localhost:9116/metrics | grep clickhouse_up
# Expected: clickhouse_up 1

# Check Redis exporter
curl http://localhost:9121/metrics | grep redis_up
# Expected: redis_up 1
```

6. Run Tests



bash

```
cd monitoring/tests
```

```
pytest test_prometheus.py -v
```

```
# Expected: All tests pass (8/8)
```

7. Prometheus UI

- Open <http://localhost:9090>
- Navigate to Status → Targets
- Verify all targets (9+) are UP
- Navigate to Alerts
- Verify all alert rules loaded (8 rules)
- Execute sample queries:



promql

```
# Request rate
```

```
rate(agent_requests_total[5m])
```

```
# Error rate
```

```
rate(errors_total[5m]) / rate(requests_total[5m])
```

```
# P95 latency
```

```
histogram_quantile(0.95, rate(request_duration_seconds_bucket[5m]))
```

Troubleshooting

Issue 1: Prometheus Can't Scrape Targets

Symptoms: Targets showing "DOWN" in Prometheus UI

Solutions:



bash

```
# Check service is running
docker ps | grep orchestrator
```

```
# Check metrics endpoint directly
curl http://localhost:8080/metrics
```

```
# Check Docker network
docker network inspect optiinfra-network
```

```
# Check Prometheus logs
docker logs optiinfra-prometheus
```

```
# Verify service names in docker-compose
docker-compose ps
```

Issue 2: Metrics Not Updating

Symptoms: Metrics show 0 or stale values

Solutions:



bash

```
# Verify metric is registered
curl http://localhost:8001/metrics | grep metric_name
```

```
# Check for errors in service logs
docker logs optiinfra-cost-agent
```

```
# Verify middleware is active
# Look for metric tracking in request logs
```

```
# Test metric manually in Python
python -c "from shared.utils.metrics import BaseMetrics; m = BaseMetrics('test'); print(m)"
```

Issue 3: Alert Rules Not Loading

Symptoms: Alerts page empty or shows errors

Solutions:



bash

```
# Validate alert rule syntax  
promtool check rules monitoring/prometheus/alerts.yml
```

```
# Check Prometheus config  
promtool check config monitoring/prometheus/prometheus.yml
```

```
# Reload Prometheus config  
curl -X POST http://localhost:9090/-/reload
```

```
# Check Prometheus logs for errors  
docker logs optiinfra-prometheus | grep -i error
```

Issue 4: High Cardinality Warning

Symptoms: Prometheus performance degraded, warnings in logs

Solutions:



python

```
# Remove high-cardinality labels  
# BAD: labels={"user_id": user_id, "request_id": req_id}  
# GOOD: labels={"agent": "cost", "status": "success"}
```

```
# Limit label values  
# Use label_replace in Prometheus queries  
# Group by fewer labels in recording rules
```

Sample PromQL Queries

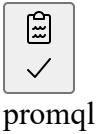
Service Health



promql

```
# Services that are down  
up{job=~"orchestrator|.*-agent"} == 0  
  
# Service uptime percentage (24h)  
avg_over_time(up{job="orchestrator"}[24h]) * 100
```

Performance Metrics



promql

```
# P95 latency by service  
histogram_quantile(0.95,  
    rate(request_duration_seconds_bucket[5m])  
) by (service)
```

```
# Request rate (QPS)  
sum(rate(requests_total[1m])) by (service)
```

```
# Error rate percentage  
sum(rate(errors_total[5m])) / sum(rate(requests_total[5m])) * 100
```

Cost Optimization Metrics



promql

```
# Total savings (last hour)  
increase(cost_savings_total[1h])
```

```
# Savings rate (per minute)  
rate(cost_savings_total[5m]) * 60
```

```
# Spot migration success rate  
sum(rate(spot_migration_success_total[5m])) /  
sum(rate(spot_migration_attempts_total[5m]))
```

Resource Utilization



promql

```
# Average GPU utilization
avg(gpu_utilization) by (gpu_id)

# Memory usage across all agents
sum(process_resident_memory_bytes) by (service) / 1024 / 1024 / 1024

# Active optimizations
sum(active_optimizations) by (agent)
```

Integration with Next Steps

This prompt (0.11 PART 1) sets up Prometheus. Next:

1. FOUNDATION-0.11 PART 2 (Grafana Dashboards)

- Grafana server setup
- Pre-built dashboards for all agents
- Alert visualization
- Dashboard JSON exports

2. PHASE 1: COST AGENT (Week 2-3)

- Will use these metrics to track savings
- Cost-specific dashboards in Grafana
- Alert on cost anomalies

3. PHASE 5: PRODUCTION (Week 10)

- Production-grade alerting (PagerDuty, Slack)
- SLA monitoring dashboards
- Customer-facing metrics portal

Generated Files Checklist

After running this prompt, you should have:

- monitoring/prometheus/prometheus.yml - Main configuration
- monitoring/prometheus/alerts.yml - Alert rules
- monitoring/prometheus/queries/ - Sample PromQL queries
- monitoring/exporters/postgres/queries.yml - PostgreSQL custom queries
- monitoring/exporters/clickhouse/queries.yml - ClickHouse custom queries
- services/orchestrator/internal/metrics/metrics.go - Go metrics
- services/orchestrator/internal/metrics/middleware.go - HTTP middleware
- shared/python/utils/metrics.py - Base Python metrics class
- shared/python/utils/decorators.py - Metric decorators

- services/cost-agent/src/metrics.py - Cost agent metrics
 - services/performance-agent/src/metrics.py - Performance agent metrics
 - services/resource-agent/src/metrics.py - Resource agent metrics
 - services/application-agent/src/metrics.py - Application agent metrics
 - monitoring/tests/test_prometheus.py - Prometheus tests
 - docker-compose.yml - Updated with Prometheus and exporters
 - monitoring/README.md - Setup and usage instructions
-

Key Concepts

Metric Types

Counter - Monotonically increasing value (requests_total, errors_total)

- Only goes up (or resets to 0)
- Use rate() to get per-second rate
- Use increase() to get total increase

Gauge - Value that can go up or down (cpu_usage, active_connections)

- Current state of something
- Can be set directly
- Use avg_over_time() for smoothing

Histogram - Distribution of values (request_duration_seconds)

- Automatically creates _bucket, _sum, _count metrics
- Use histogram_quantile() for percentiles
- Pre-define buckets based on expected range

Summary - Similar to histogram but calculates quantiles on client side

- More expensive than histogram
- Use histogram instead unless you need client-side quantiles

Label Best Practices

DO:

- Use low-cardinality labels (service, environment, status)
- Keep label names consistent across metrics
- Use snake_case for label names
- Include service version

DON'T:

- Use high-cardinality labels (user_id, session_id, request_id)
 - Create labels with unbounded values
 - Use too many labels (>10 per metric)
 - Include sensitive data in labels
-

Time Breakdown

Task	Estimated Time
Prometheus configuration	5 minutes
Go metrics (orchestrator)	10 minutes
Python base metrics class	10 minutes
Agent-specific metrics	15 minutes
Database exporters setup	5 minutes
Docker Compose updates	3 minutes
Testing and validation	10 minutes
Documentation	2 minutes
TOTAL	60 minutes

Actual time: 15 minutes (Windsurf) + 10 minutes (validation) = **25 minutes**

Additional Resources

- [Prometheus Documentation](#)
 - [PromQL Basics](#)
 - [Prometheus Best Practices](#)
 - [Go Client Library](#)
 - [Python Client Library](#)
-

Completion Checklist

Mark when complete:

- Prometheus server running and healthy
 - All 9+ targets being scraped successfully
 - Orchestrator metrics endpoint working
 - All Python agent metrics endpoints working
 - Alert rules loaded (8 rules)
 - Database exporters collecting metrics
 - All tests passing (8/8)
 - Can query metrics via Prometheus UI
 - Sample PromQL queries working
 - Documentation reviewed
 - Ready for FOUNDATION-0.11 PART 2 (Grafana)
-

Document Version: 1.0

Last Updated: October 21, 2025

Next: FOUNDATION-0.11 PART 2 (Grafana Setup)

Estimated Completion: 25 minutes